DEVELOPMENTARC®

React In-Depth:

Developing production applications with React

A free open source GitBook

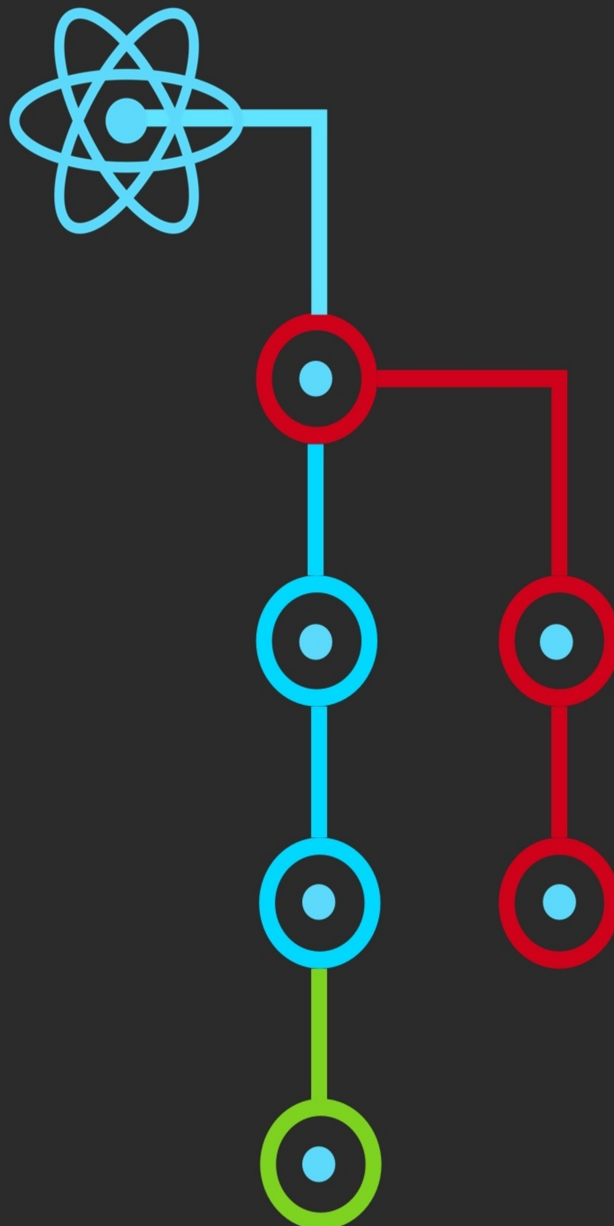


Table of Contents

React In-depth: An exploration of UI development

> From: DevelopmentArc/react-indepth

At DevelopmentArc® we have a love affair with UI application development. We truly enjoy the exploration of technologies and frameworks. This process helps us better understand the end-goals and complexity of the application stack.

We discovered early on the need to apply true software development principles to the user interface. This discovery started a long time ago with technologies like Authorware/Toolbook and Director. Quickly we moved to HTML 1.0 and began to build systems with JavaScript frameworks like Prototype.js. With the release of the Flex framework, our focus again shifted to a new platform, Flash.

For a majority, Flash was a negative experience and UI technology. While many arguments are true, we found Flash to be the most advanced cross-platform rendering system available at the time. Flex gave us the application framework necessary to build the large and complex applications clients were requesting. All the while, Flex lent itself to the management of large and diverse teams. Our obsession with Flex included a deep understand of the application and component life cycles, resulting in a 90-page white paper. The paper is still available and continues to be referenced today. With the collapse of the Flash Platform and Flex, we find ourselves back in HTML and JavaScript world. This time our obsession is React.js

Throughout the years we have continued to try and push the boundaries of UI technologies. Starting, in late 2014, we began initial research and then full adoption of React.js for our UI layer in for web applications.

The initial process of moving to React was a blend of excitement and at times, pure frustration. React brings in both existing UI paradigms and also new patterns that can take a bit of time to adjust your own mind-set to. Once we had fully grokked React, we found that it has opened the possibilities for our current and future projects.

Our goal with this GitBook is to document our process, share our research and try to organize our experiences into a single living document. Too be honest, this is a pretty lofty goal. We may not achieve this goal, but hopefully we can get some helpful thoughts down.

We have found the longer you write code, the more you learn and then shortly forget. This means that our writings are just as much for ourselves as for others. With that in mind we hope that as this document grows it will help you, just as much as it helped us to put our own

thoughts down.

James & Aaron

[WIP] React From The Inside Out

This section is currently being researched

In the meantime, there are ton of getting started resources already available. If you are new to React we recommend spending some time looking at these fantastic resources:

- React Official Site
- Awesome React - A comprehensive list of resources
- Learning React, Getting Started - Basic intro article

From a book development perspective, our first focus will be on the Life Cycle chapters and we will circle back to this section soon.

Guiding Principals to learning React

Unlike most intro books, we want to have different goals for our Basics Section. We will dig into the underpinnings of React and look at how to create applications from the bottom up. We feel that understanding the internals of a UI system helps drive development choices. We plan to explore React from this guiding principal.

The React Life Cycle

One of the defining factors of a life form is its life cycle. The common path is Birth, Growth into maturity and then the inevitable Death. UI applications often follow a similar path. When the application is first started, we consider this Birth. The users interacts with application, which is Growth. Eventually, the application is closed or navigated away from, leading to Death.

Within the application, elements also follow this pattern. In the world of React, these elements are our Components. The Component life cycle is a continuous process, which occurs throughout the overall life of our application. Understanding this process can lead to faster and consistent development, easier optimization and improved overall application health.

Life cycle phases in React components

Not all UI systems enable a life cycle pattern. This doesn't mean that a system is better or worse if a life cycle is or isn't implemented. All a life cycle does is provide a specific order of operation and a series of hooks to tie into said system. The React life cycle follows the common Birth, Growth, and Death flow. The React team has provided a series of methods you can implement/override to tap into the process.

Phase 1: Birth / Mounting

The first phase of the React Component life cycle is the Birth/Mounting phase. This is where we start initialization of the Component. At this phase, the Component's `props` and `state` are defined and configured. The Component and all its children are mounted on to the Native UI Stack (DOM, UIView, etc.). Finally, we can do post-processing if required. The Birth/Mounting phase only occurs once.

Phase 2: Growth / Update

The next phase of the life cycle is the Growth/Update phase. In this phase, we get new `props` , change `state` , handle user interactions and communicate with the component hierarchy. This is where we spend most of our time in the Component's life. Unlike Birth or Death, we repeat this phase over and over.

Phase 3: Death / Unmount

The final phase of the life cycle is the Death/Unmount phase. This phase occurs when a component instance is unmounted from the Native UI. This can occur when the user navigates away, the UI page changes, a component is hidden (like a drawer), etc. Death occurs once and readies the Component for Garbage Collection.

Next Up: Life Cycle Methods Overview

React Life Cycle Methods Overview

The React development team provides a series of hooks we can tap into at each phase of the life cycle. These method hooks inform us of where the Component is in the life cycle and what we can and cannot do.

Each of the life cycle methods are called in a specific order and at a specific time. The methods are also tied to different parts of the life cycle. Here are the methods broken down in order and by their corresponding life cycle phase [1]:

Birth / Mounting

1. Initialize / Construction
2. `getDefaultProps()` *(React.createClass)* or `MyComponent.defaultProps` *(ES6 class)*
3. `getInitialState()` *(React.createClass)* or `this.state = ...` *(ES6 constructor)*
4. `componentWillMount()`
5. `render()`
6. Children initialization & life cycle kickoff
7. `componentDidMount()`

Growth / Update

1. `componentWillReceiveProps()`
2. `shouldComponentUpdate()`
3. `componentWillUpdate()`
4. `render()`
5. Children Life cycle methods
6. `componentDidUpdate()`

Death / Unmount

1. `componentWillUnmount()`
2. Children Life cycle methods
3. Instance destroyed for Garbage Collection

The order of these methods are strict and called as defined above. Most of the time is spent in the Growth/Update phase and those methods are called many times. The Birth and Death methods will only be called once.

Next Up: Birth/Mounting in-depth

---

[1] *Most of the methods are the same if you use either `React.createClass` or use ES6 classes, such as `class MyComponent extends React.Component`. A few are different, mainly around how instantiation/creation occurs. We will call these differences out throughout the chapter.*

Birth/Mounting In-depth

A React Component kicks off the life cycle during the initial application ex: `ReactDOM.render()`. With the initialization of the component instance, we start moving through the Birth phase of the life cycle. Before we dig deeper into the mechanics of the Birth phase, let's step back a bit and talk about what this phase focuses on.

The most obvious focus of the birth phase is the initial configuration for our Component instance. This is where we pass in the `props` that will define the instance. But during this phase there are a lot more moving pieces that we can take advantage of.

In Birth we configure the default `state` and get access to the initial UI display. It also starts the mounting process for children of the Component. Once the children mount, we get first access to the Native UI layer[1] (DOM, UIView, etc.). With Native UI access, we can start to query and modify how our content is actually displayed. This is also when we can begin the process of integrating 3rd Party UI libraries and components.

Components vs. Elements

When learning React, many developers have a common misconception. At first glance, one would assume that a mounted instance is the same as a component class. For example, if I create a new React component and then `render()` it to the DOM:

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyComponent extends React.Component {
  render() {
    return <div>Hello World!</div>;
  }
};

ReactDOM.render(<MyComponent />, document.getElementById('mount-point'));
```

The initial assumption is that during `render()` an instance of the `MyComponent` class is created, using something like `new MyComponent()`. This instance is then passed to render. Although this sounds reasonable, the reality of the process is a little more involved.

What is actually occurring is the JSX processor converts the `<MyComponent />` line to use `React.createElement` to generate the instance. This generated Element is what is passed to the `render()` method:

```
// generated code post-JSX processing
ReactDOM.render(
  React.createElement(MyComponent, null), document.getElementById('mount-point')
);
```

A React Element is really just a description[2] of what will eventually be used to generate the Native UI. This is a core, pardon the pun, *element* of virtual DOM technology in React.

> The primary type in React is the ReactElement. It has four properties: type, props, key and ref. It has no methods and nothing on the prototype.
>
> -- https://facebook.github.io/react/docs/glossary.html#react-elements

The Element is a lightweight object representation of what will become the component instance. If we try to access the Element thinking it is the Class instance we will have some issues, such as availability of expected methods.

So, how does this tie into the life cycle? These descriptor Elements are essential to the creation of the Native UI and are the catalyst to the life cycle.

The First `render()`

To most React developers, the `render()` method is the most familiar. We write our JSX and layout here. It's where we spend a lot of time and drives the layout of the application. When we talk about the first `render()` this is a special version of the `render()` method that mounts our entire application on the Native UI.

In the browser, this is the `ReactDOM.render()` method. Here we pass in the root Element and tell React where to mount our content. With this call, React begins processing the passed Element(s) and generate instances of our React components. The Element is used to generate the type instance and then the `props` are passed to the Component instance.

This is the point where we enter the Component life cycle. React uses the `instance` property on the Element and begins construction.

Next Up: Initialization & Construction

---

[1] The Native UI layer is the system that handles UI content rendering to screen. In a browser, this is the DOM. On device, this would be the UIView (or comparable). React handles the translation of Component content to the native layer format.

[2] Dan Abramov chimed in with this terminology on a StackOverflow question. http://stackoverflow.com/a/31069757

Initialization & Construction

During the initialization of the Component from the Element, the `props` and `state` are defined. How these values are defined depends on if you are using `React.createClass()` or `extend React.Component`. Let's first look at `props` and then we will examine `state`.

Default Props

As we mentioned earlier, the Element instance contains the current `props` that are being passed to the Component instance. Most of the time, all the available `props` on the Component are not required. Yet, some times we do need to have values for all the `props` for our Component to render correctly.

For example, we have a simple component that renders a name and age.

```
import React from 'react';

export default class Person extends React.Component {
  render() {
    return (
      <div>{ this.props.name } (age: { this.props.age })</div>
    );
  }
}
```

In our case, we expect two props to be passed in: `name` and `age`. If we want to make `age` optional and default to the text 'unknown' we can take advantage of React's default props.

For ES6 Class

```
import React from 'react';

class Person extends React.Component {
  render() {
    return (
      <div>{ this.props.name } (age: { this.props.age })</div>
    );
  }
}

Person.defaultProps = { age: 'unknown' };

export default Person;
```

For createClass (ES6/ES5/CoffeeScript, etc.)

```
var Person = React.createClass({
  getDefaultProps: function() {
    return ({ age: 'unknown' });
  },

  render: function() {
    return (
      <div>{ this.props.name } (age: { this.props.age })</div>
    );
  }
});
```

The result of either process is the same. If we create a new instance without setting the age prop ex: `<Person name="Bill" />`, the component will render `<div>Bill (age: unknown)</div>`.

React handles default props by merging the passed props object and the default props object. This process is similar to `Object.assign()` or the Lodash/Underscore `_.assign()` method. The default props object is the target object and the passed props is the source:

```
// React library code to extract defaultProps to the Constructor
if (Constructor.getDefaultProps) {
   Constructor.defaultProps = Constructor.getDefaultProps();
}

// psuedo code (as an example)
this.props = Object.assign(Constructor.defaultProps, elementInstance.props);
```

In the React code snippet, React checks the underlying Class instance to see if it defines `getDefaultProps()` and uses this to set the values. When using ES6 classes we just define it on the class itself. Any property defined on the `passedProps` value is applied/overridden to the property in the default props object.

`null` vs. `undefined` props

When using default props, it is important to understand how the React merge process works. Often, we are generating props dynamically based on application state (Flux, Redux, Mobx, etc.). This means that we can sometimes generate `null` values and pass this as the prop.

When assigning default props, the React object merge code sees `null` as a defined value.

```
<Person name="Bob" age={ null } />
```

Because `null` is a defined value our Component would render this as `<div>Bob (age:)</div>` instead of rendering *unknown*. But, if we pass in `undefined` instead of `null`, React treats this as undefined (well yeah, obviously) and we would render *unknown* as expected.

Keep this in mind when defining default props, because tracing down a `null` value can be tricky in larger application.

Initial State

Once the final props are defined (passed w/ defaults), the Component instance configures the initial `state`. This process occurs in the construction of the instance itself. Unlike props, the Component state is an internal object that is not defined by outside values.

To define the initial `state` depends on how you declare your Component. For ES6 we declare the state in the constructor. Just like `defaultProps`, the initial state takes an object.

For ES6 Class

```
import React from 'react';

class Person extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return (
      <div>{ this.props.name } (age: { this.props.age })</div>
    );
  }
}

Person.defaultProps = { age: 'unknown' };

export default Person;
```

For `React.createClass` Components, there is a helper method called `getInitialState()` which returns the state object. This method is called during setup to set the state on the instance.

For createClass (ES6/ES5/CoffeeScript, etc.)

```
var Person = React.createClass({
  getDefaultProps: function() {
    return ({ age: 'unknown' });
  },

  getInitialState: function() {
    return ({ count: 0 });
  },

  render: function() {
    return (
      <div>{ this.props.name } (age: { this.props.age })</div>
    );
  }
});
```

State defaults

It is important to keep in mind that if we do not define a state in the constructor/getInitialState then the state will be `undefined`. Because the state is `undefined` and not an empty Object (`{}`), if you try to query the state later on this will be an issue.

In general, we want to set a default value for all our state properties. There are some edge cases where the initial value for the state property may be `null` or `undefined`. If this state happens to be only state property, it may be tempting to skip setting a default state. But, if our code tries to access the property you will get an error.

```
  class Person extends React.Component {
   render() {
     // This statement will throw an error
     console.log(this.state.foo);
     return (
       <div>{ this.props.name } (age: { this.props.age })</div>
     );
   }
 }
```

The log statement fails because `this.state` is undefined. When we try to access `foo` we will get a *TypeError: Cannot read property 'foo' of null*. To solve this we can either set the default state to `{}` or, to have a clearer intention, set it to `{ foo: null }`.

Next Up: Pre-mounting with `componentWillMount()`

Pre-mounting with `componentWillMount()`

Now that the props and state are set, we finally enter the realm of Life Cycle methods. The first true life cycle method called is `componentWillMount()`. This method is only called one time, which is before the initial render. Since this method is called before `render()` our Component will not have access to the Native UI (DOM, etc.). We also will not have access to the children `refs`, because they are not created yet.

The `componentWillMount()` is a chance for us to handle configuration, update our state, and in general prepare for the first render. At this point, props and initial state are defined. We can safely query `this.props` and `this.state`, knowing with certainty they are the current values. This means we can start performing calculations or processes based on the prop values.

Person.js [1]

```
import React from 'react';
import classNames from 'classnames';

class Person extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mode: undefined } ;
  }

  componentWillMount() {
    let mode;
    if (this.props.age > 70) {
      mode = 'old';
    } else if (this.props.age < 18) {
      mode = 'young';
    } else {
      mode = 'middle';
    }
    this.setState({ mode });
  }

  render() {
    return (
      <div className={ classNames('person', this.state.mode) }>
        { this.props.name } (age: { this.props.age })
      </div>
    );
  }
}

Person.defaultProps = { age: 'unknown' };

export default Person;
```

In the example above we call `this.setState()` and update our current state before render. If we need state values on calculations passed in `props`, this is where we should do the logic.

Other uses for `componentWillMount()` includes registering to global events, such as a Flux store. If your Component needs to respond to global Native UI events, such as `window` resizing or focus changes, this is a good place to do it[2].

Next Up: Component `render()`

---

[1] In our example above, we are using the `classNames()` library, which was originally included as a React Addon. However, the feature has been removed from React and moved to its own library for use with or without React.

[2] It's important to remember that many Native UI elements do not exist at this point in the life cycle. That means we need to stick to very high-level/global events such as `window` or `document`.

Component `render()`

Now that we have pre-configured our Component, we enter the first rendering of our content. As React developers, the `render()` method is the most familiar. We create Elements (generally via JSX) and return them. We access the Component `this.props` and `this.state` and let these values derive how content should be generated. When we access `this.state`, any changes we made during `componentWillMount()` are fully applied.

Unlike any other method in the Life Cycle, `render()` is the one method that exists across multiple life cycle phases. It occurs here in Birth and it is where we spend a lot of time in Growth.

In both cases, we have the core principle of keeping `render()` a pure method. What does that mean? That means we shouldn't call `setState()`, query the Native UI or anything else that can mutate the existing state of the application. The reason why is if we do this kind of interaction in `render()`, then it will kickoff another render pass. Which once again, triggers `render()` which then does the same thing... infinitely.

The React development mode[1] is generally great at catching these kinds of errors and will yell at you if you do them. For example, if we did something silly like this

```
render() {
  // BAD: Do not do this!
  this.setState({ foo: 'bar' });
  return (
    <div className={ classNames('person', this.state.mode) }>
      { this.props.name } (age: { this.props.age })
    </div>
  );
}
```

React would log out the following statement:

> Warning: setState(...): Cannot update during an existing state transition (such as within `render`). Render methods should be a pure function of props and state.

Native UI access in `render()` is often fatal

React will also warn you if you try to access the Native UI elements in the render pass.

```
render() {
  // BAD: Don't do this either!
  let node = ReactDOM.findDOMNode(this);
  return (
    <div className={ classNames('person', this.state.mode) }>
      { this.props.name } (age: { this.props.age })
    </div>
  );
}
```

> VM943:45 Warning: Person is accessing getDOMNode or findDOMNode inside its render(). render() should be a pure function of props and state. It should never access something that requires stale data from the previous render, such as refs. Move this logic to componentDidMount and componentDidUpdate instead.

In the above example, it may seem safe since you are just querying the node. But, as the warning states, we might be querying potentially old data. But in our case, during the Birth phase, this would be a fatal error.

> Uncaught Invariant Violation: findComponentRoot(..., .0): Unable to find element. This probably means the DOM was unexpectedly mutated (e.g., by the browser), usually due to forgetting a <tbody> when using tables, nesting tags like <form>, <p>, or <a>, or using non-SVG elements in an <svg> parent. Try inspecting the child nodes of the element with React ID `Person`.

This is one of those cases where the React error doesn't clearly point to the cause of the problem. In our case we didn't modify the DOM, so it feels like an unclear and potentially misleading error. This kind of error can cause React developers a lot of pain early on. Because we instinctually look for a place where we are changing the Native UI.

The reason we get this error is because during the first render pass the Native UI elements we are trying to access do not exist yet. We are essentially asking React to find a DOM node that doesn't exist. Generally, when `ReactDOM` can't find the node, this is because something or someone mutated the DOM. So, React falls back to the most common cause.

As you can see, having an understanding of the Life Cycle can help troubleshoot and prevent these often un-intuitive issues.

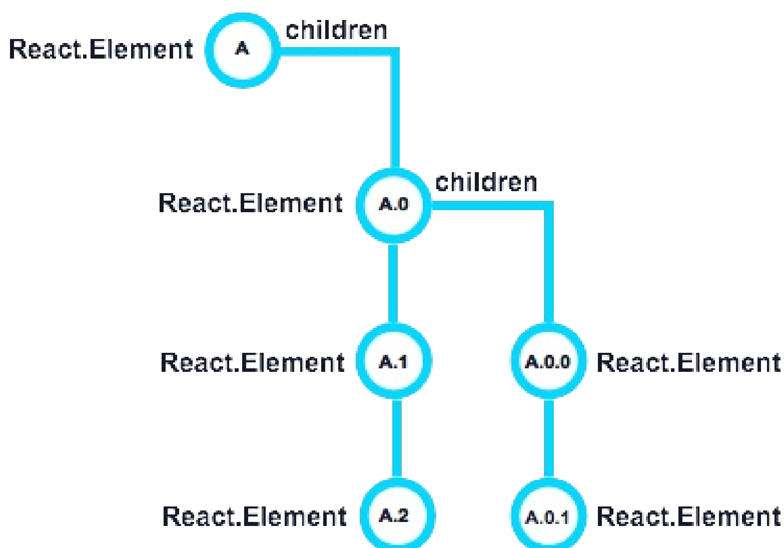
Up Next: Managing Children Components and Mounting

---

[1] These warnings come out of the development mode in React. You can also use the React Development Tools to help debug and explore React components.

Managing Children Components and Mounting

Now that we have completed the first render pass, our `render()` method returns a single React Element. This Element may have children elements. Those children may also have children, and so on.



With the potential for an *n* depth tree of Elements, each of the Elements need to go through their own entire life cycle process. Just like the parent Element, React creates a new instance for each child. They go through construction, default props, initial state, `componentWillMount()` and `render()`. If the child has children, the process starts again...all the way down.

One of the most powerful concepts in React is the ability to easily compose complex layout through nesting of children. It is encouraged to keep your Components as *'dumb'* as possible. The idea is to only have container[1] components managing higher level functionality.

Because this is the preferred way of development, this means we will have a lot of smaller components that also have their own life cycle. Keep this in mind as we continue through the life cycle, because every Component will follow the same pattern.

Up Next: Post-Mount with `componentDidMount()`

---

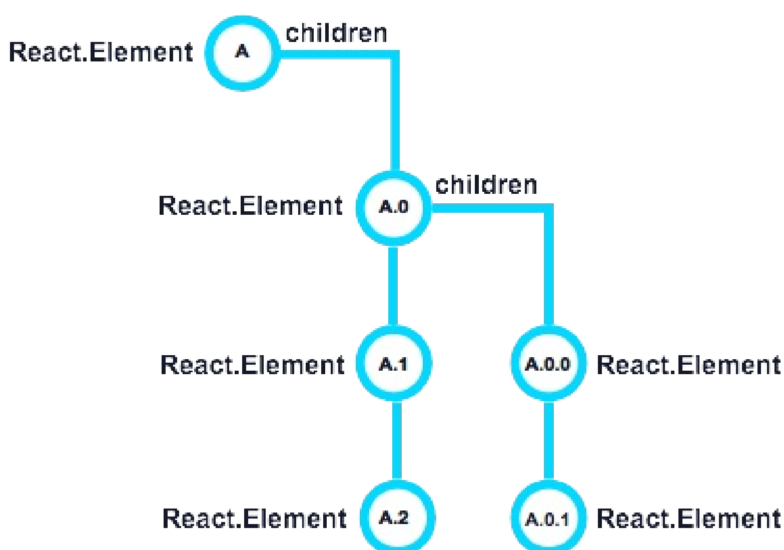
[1] See Presentational and Container Components by Dan Abramov for more details

Post-mount with `componentDidMount()`

The last step in the Birth/Mount life cycle phase is our post-mount access via `componentDidMount()`. This method is called once all our children Elements and our Component instances are mounted onto the Native UI. When this method is called we now have access to the Native UI (DOM, UIView, etc.), access to our children `refs` and the ability to *potentially* trigger a new render pass.

Understanding call order

Similar to `componentWillMount()`, `componentDidMount()` is only called one time. Unlike our other Birth/Mount methods, where we start at the top and work down, `componentDidMount()` works from the bottom up. Let's consider the following Component/Element Tree again:



When we begin the Birth phase, we process `render()` in this order:

```
A -> A.0 -> A.0.0 -> A.0.1 -> A.1 -> A.2.
```

With `componentDidMount()` we start at the end and work our way back.

```
A.2 -> A.1 -> A.0.1 -> A.0.0 -> A.0 -> A
```

By walking backwards, we know that every child has mounted and also run its own `componentDidMount()`. This guarantees the parent can access the Native UI elements for itself and its children.

Let's consider the following three components and their call order.

GrandChild.js

```
/**
 * GrandChild
 * It logs the componentDidMount() and has a public method called value.
 */
import React from 'react';
import ReactDOM from 'react-dom';

export default class GrandChild extends React.Component {

  componentDidMount() {
    console.log('GrandChild did mount.');
  }

  value() {
    return ReactDOM.findDOMNode(this.refs.input).value;
  }

  render() {
    return (
      <div>
        GrandChild
        <input ref="input" type="text" defaultValue="foo" />
      </div>
    );
  }
}
```

Child.js

```
/*
 * Child
 * It logs the componentDidMount() and has a public method called value,
 * which returns the GrandChild value.
 */
import React from 'react';
import GrandChild from './GrandChild';

export default class Child extends React.Component {

  componentDidMount() {
    console.log('Child did mount.');
  }

  value() {
    return this.refs.grandChild.value();
  }

  render() {
    return (
      <div>
        Child
        <GrandChild ref="grandChild" />
      </div>
    );
  }
}
```

Parent.js

```
/*
 * Parent
 * It logs the componentDidMount() and then logs the child value()
 * method.
 */
import React from 'react';
import Child from './Child';

export default class Parent extends React.Component {

  componentDidMount() {
    console.log('Parent did mount.');
    console.log('Child value:', this.refs.child.value());
  }

  render() {
    return (
      <div>
        Parent
        <Child ref="child" />
      </div>
    );
  }
}
```

When we mount `<Parent />` in our application we get the following in the browser console:

```
GrandChild did mount.
Child did mount.
Parent did mount.
Child value: foo
```

As you can see, the GrandChild's `componentDidMount()` was called first, followed by Child and then Parent. Because we are now mounted on the DOM and our children are created, the Parent can access its `refs` and the GrandChild can access its own DOM nodes.

Useful Tasks

The `componentDidMount()` method can be a helpful heavy lifter for our Components. One of the most common tasks is interacting with the Native UI. Unlike `componentWillMount()` or `render()` we can now fully interact with the Native stack.

For example, we may need to make changes to our current state based on how the Native UI laid out our content. We may need to figure out the current width/height of our children or our own instance. This is especially helpful in the browser where CSS layout drives a lot of our DOM calculations.

Another useful task is setting up 3rd party UIs. For example, if we wanted to use a library like C3.js or the Date Range Picker, this is where we would initialize our UI libraries.

Chart.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import c3 from 'c3';

export default class Chart extends React.Component {

  componentDidMount() {
    this.chart = c3.generate({
      bindto: ReactDOM.findDOMNode(this.refs.chart),
      data: {
        columns: [
          ['data1', 30, 200, 100, 400, 150, 250],
          ['data2', 50, 20, 10, 40, 15, 25]
        ]
      }
    });
  }

  render() {
    return (
      <div ref="chart"></div>
    );
  }
}
```

In the above example, we leverage `componentDidMount()` to generate our chart, bind it to the DOM using `refs` and then pass in data.

When integrating 3rd party libraries, we often need to bind to events, such as the user interacting with the Chart. This is where we would set up our listeners post-library initialization. We can also add more global listeners here, if we did not want to setup the listeners in the `componentWillMount()` call.

Starting another render pass [1]

There are some unique situations where we may have a second render immediately after Birth/Mount. This is not a common situation and generally occurs when we have to change our current state based on the Native UI Layout. This could be calculating dynamic row height or column widths in a data table. It could be having to re-position the component's children based on how they are sized the first time.

If you require this kind of functionality, you have the ability to call `this.setState()` or `forceUpdate()` in your `componentDidMount()`. If you change state or force an update (more on this feature later), your Component will begin another render pass and enter the Growth/Update Phase. Because `componentDidMount()` is called only once, we don't have to worry about this method causing an infinite loop. But, this process can lead to issues down the road if you do not take the time to walk through all the potential ramifications of multiple renders.

Up Next: Growth/Update Phase In-Depth

1

[1] Multiple render passes opens the door for serious performance issues. Proceed with extreme caution!

Growth/Update In-depth

Once our Component is mounted in the Birth phase, we are prepped and ready for the Growth/Update phase. The Growth phase is where a Component spends most of its time. Here we get data updates, act on user or system actions and provide the overall experience for our application.

The Growth phase is triggered in three different ways: changing of `props`, changing of `state` or calling `forceUpdate()`. The changes that are made affect how the Update phase is managed. We will discuss each of these changes in depth as we walk through the entire Growth process.

In this Section, we will dive into the different methods. We'll examine the order of the methods called and how they affect the overall process. We will also discuss what tasks are best handled during each method and discuss application optimization.

Starting Update: Changing Props

As mentioned earlier, we have three ways to start the Growth/Update phase. The first way is when the component's `props` update. This occurs when either the root Element (ex: `ReactDOM.render(<MyComponent data={ dataVaule } />, ...)` has the `props` value changed or the parent of the Component's `prop` changes.

From a Component's instance perspective (such as `<Person name="Bill" />`) the passed in props are immutable. In other words, the Person instance cannot update the value name internally. In fact, if you try you will get an Error in React.

```
render() {
  // BAD: DON'T DO THIS!
  this.props.name = 'Tim';
  return (
    <div className={ classNames('person', this.state.mode) }>
      { this.props.name } (age: { this.props.age })
    </div>
  );
}
```

> TypeError: Cannot assign to read only property 'name' of object '#<Object>'

Because props are immutable by the Component itself, the parent must provide the new values. When new props are passed in via root or the parent, this starts the Update phase.

Starting Update: `setState()`

Similar to changing `props`, when a Component changes its state value[1] via `this.setState()`, this also triggers a new Update phase. For a lot of React developers, the first major (and to be honest ongoing) challenge is managing state in Components. State itself can be a controversial topic in the community. Many developers avoid state at all cost. Other systems, such as MobX, are in essence trying to replace it. Many uses of state can fall into different anti-patterns, such as transferring `props` into `state`[2].

Fundamentally, state can be a tricky and confusing topic. When do we use state? What data should or shouldn't be stored in state? Should we even use state at all? To be honest, this is a topic that we are still trying to grapple with ourselves.

Keeping that in mind, it is still important to understand how state works in React. We will continue to discuss state in-depth and how the mechanics work. We will try to share best practices that we have found, but in general what is good today will probably be bad tomorrow.

The asynchronicity of state

Before we move on to the final way to start an update, we should talk a little about how state is managed in the internals of React. When developers first start using `setState()` there is an assumption that when you call `this.state` the values applied on the set will be available. This is not true. The `setState()` method should be treated as an asynchronous process[3]. So how does `setState()` work?

When we call `setState()` this is considered a partial state change. We are not flushing/replacing the entire state, just updating part(s) of it. React uses a queuing system[4] to apply the partial state change. Because we can set the state multiple times in a method chain, a change queue is constructed to manage all the various updates. Once the state change is added to the queue, React makes sure the Component is added to the dirty queue. This dirty queue tracks the Component instances that have changed. Essentially, this is what tells React which Components need to enter the Update phase later.

When working with state, it is very important to keep this in mind. A common error is to set state in one method and then later in the same synchronous method chain try to access the state value. This can sometimes cause tricky bugs, especially if you expose state values via public methods on your Component, such as `value()`. We will talk later about when `this.state` is finalized.

Starting Update: `forceUpdate`

There is one more way to kick off an Update phase. There is a special method on a Component called `forceUpdate()`. This does exactly what you think, it forces the Component into an Update phase. The `forceUpdate()` method has some specific ramifications about how the life cycle methods are processed and we will discuss this in-depth later on.

Up Next: Updating and `componentWillReceiveProps()`

---

[1] With Component state, we consider this internal functionality. In theory, we can access and even edit state outside of the instance but this is an anti-pattern. Accessing a Component's state from outside injects a lot of fragility into the system (pathing dependency, changing of internal values, etc.). Only a Component instance should `setState()` on itself.

[2] Even though moving `props` to `state` is considered an anti-pattern there are a few use cases. The most common is having a `defaultValue` prop that becomes the internal `value` in state. We see this pattern with most Form elements in React; although there is a strong movement to get away from this and work with only Controlled Components.

[3] The React code comments recommend that *"... You should treat `this.state` as immutable. There is no guarantee that `this.state` will be immediately updated, so accessing `this.state` after calling [the setState] method may return the old value."*

[4] React internal method `enqueueSetState()` to be exact.

Updating and `componentWillReceiveProps()`

Now that we have discussed starting an Update, let's dive into the Update life cycle methods. The first method available to us is `componentWillReceiveProps()` . This method is called when `props` are passed to the Component instance. Let's dig a little deeper into what this means.

Passing `props`

The most obvious example is when new `props` are passed to a Component. For example, we have a Form Component and a Person Component. The Form Component has a single `<input />` that allows the user to change the name by typing into the input. The input is bound to the `onChange` event and sets the state on the Form. The state value is then passed to the Person component as a `prop` .

Form.js

```
import React from 'react';
import Person from './Person';

export default class Form extends React.Component {
  constructor(props) {
    super(props);
    this.state        = { name: '' } ;
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({ name: event.currentTarget.value });
  }

  render() {
    return (
      <div>
        <input type="text" onChange={ this.handleChange } />
        <Person name={ this.state.name } />
      </div>
    );
  }
}
```

Any time the user types into the `<input />` this begins an Update for the Person component. The first method called on the Component is `componentWillReceiveProps(nextProps)` passing in the new `prop` value. This allows us to compare the incoming `props` against our current `props` and make logical decisions based on the value. We can get our current props by calling `this.props` and the new value is the `nextProps` argument passed to the method.

Updating State

So why do we need `componentWillReceiveProps` ? This is the first hook that allows us to look into the upcoming Update. Here we could extract the new props and update our internal state. If we have a state that is a calculation of multiple props, we can safely apply the logic here and store the result using `this.setState()` .

> Use this as an opportunity to react to a prop transition before render() is called by updating the state using this.setState(). The old props can be accessed via this.props. Calling this.setState() within this function will not trigger an additional render.
>
> -- https://facebook.github.io/react/docs/component-specs.html#updating-componentwillreceiveprops

Props may not change

A word of caution with `componentWillReceiveProps()` . Just because this method was called, does not mean the value of props has changed.

> To understand why, we need to think about what could have happened. The data could have changed between the initial render and the two subsequent updates ... React has no way of knowing that the data didn’t change. Therefore, React needs to call `componentWillReceiveProps` , because the component needs to be notified of the new props (even if the new props happen to be the same as the old props).
>
> -- See (A => B) !=> (B => A)

The core issue with `props` and `componentWillReceiveProps()` is how JavaScript provides mutable data structures. Let's say we have a prop called `data` and data is an Array.

```
// psuedo code
this.setState({ data: [1, 2, 3] });

<MyComponent data={ this.state.data } />
```

If somewhere in our app, a process updates the data array via `push()` , this changes the content of the data Array. Yet, the Array itself is the same instance. Because it is the same instance, React can not easily determine if the internal data has changed. So, to prevent a lot of issues, or having to do deep comparisons, React will push the same props down.

With this being said, `componentWillReceiveProps()` allows us to check and see if new props are coming in and we can make choices based on the data. We just need to make sure we never assume the props are different in this method. Be sure to read the great post (A => B) !=> (B => A) by Jim Sproch.

Skipping this method

Unlike the other methods in the Mounting phase, not all our Update phase methods are called every time. For example, we will skip `componentWillReceiveProps()` if the Update is triggered by just a state change. Going back to our Form.js example above:

```
  // ...
  handleChange(event) {
     this.setState({ name: event.currentTarget.value });
   }

   render() {
     return (
       <div>
         <input type="text" onChange={ this.handleChange } />
         <Person name={ this.state.name } />
       </div>
     );
   }
 // ...
```

When the user types in the `<input />` we trigger a `setState()` method. This will trigger an Update phase in our Form Component and the Person Component. For our Form Component, we did not receive new props, so `componentWillReceiveProps()` will be skipped.

Up Next: Using `shouldComponentUpdate()

Using `shouldComponentUpdate()`

The next method in the Update life cycle is `shouldComponentUpdate()`. This method allows your Component to exit the Update life cycle if there is no reason to apply a new render. Out of the box, the `shouldComponentUpdate()` is a no-op that returns `true`. This means every time we start an Update in a Component, we will re-render.

If you recall, React does not deeply compare `props` by default. When `props` or `state` is updated React assumes we need to re-render the content. But, if the `props` or `state` have not changed, should we really be re-rendering?

Preventing unnecessary renders

The `shouldComponentUpdate()` method is the first real life cycle optimization method that we can leverage in React. We can look at our current and new `props` & `state` and make a choice if we should move on. React's PureRenderMixin does exactly this. It checks the current props and state, compares it to the next props and state and then returns `true` if they are different, or `false` if they are the same.

```
/**
 * Performs equality by iterating through keys on an object and returning false
 * when any key has values which are not strictly equal between the arguments.
 * Returns true when the values of all keys are strictly equal.
 */
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (objA === objB) {
    return true;
  }

  if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false;
  }

  var keysA = Object.keys(objA);
  var keysB = Object.keys(objB);

  if (keysA.length !== keysB.length) {
    return false;
  }

  // Test for A's keys different from B.
  var bHasOwnProperty = hasOwnProperty.bind(objB);
  for (var i = 0; i < keysA.length; i++) {
    if (!bHasOwnProperty(keysA[i]) || objA[keysA[i]] !== objB[keysA[i]]) {
      return false;
    }
  }

  return true;
}

function shallowCompare(instance, nextProps, nextState) {
  return (
    !shallowEqual(instance.props, nextProps) ||
    !shallowEqual(instance.state, nextState)
  );
}

var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  },
};
```

The above code is extracted from the React addon/source[1]. The mixin defines the `shouldComponentUpdate(nextProps, nextState)` and compares the instance's `props` against the `nextProp` and the `state` against the `nextState`.

Mutability and pure methods

It is important to note that the `shallowCompare` method simply uses `===` to check each instance. This is why the React team calls the mixin *pure*, because it will not properly check against mutable data.

Let's think back to our `data` props Array example where we use `push()` to add a new piece of data onto the Array.

```
// psuedo code
this.setState({ data: [1, 2, 3] });

<MyComponent data={ this.state.data } />
```

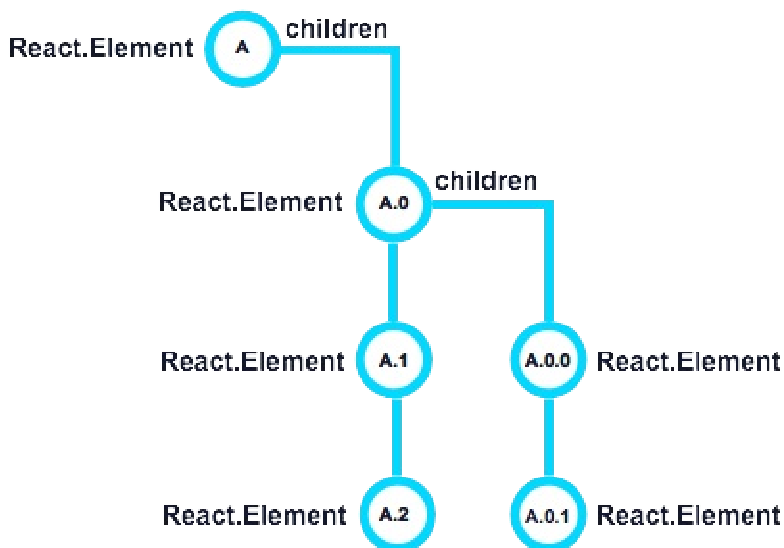
The `shallowCompare` will see the current `props.data` as the same instance as the `nextProps.data` (`props.data === nextProps.data`) and therefore not render an update. Since we mutated the `data` Array, our code is **not** considered to be *pure*.

This is why systems like Redux requires pure methods for reducers. If you need to change nested data you have to clone the objects and make sure a new instance is always returned. This allows for `shallowCompare()` to see the change and update the component.

Other ways to handle this is to use an immutable data system, such as Immutable.js. These data structures prevent developers from accidentally mutating data. By enforcing immutable data structures, we can leverage `shouldComponentUpdate()` and have it verify that our `props` and `state` have changed[2].

Stop renders at the source

If you recall our nested Component structure:



By default, if an Update is triggered in **A**, then all the other children will also go through their updates. This can easily cause a performance issue, because now we have many Components going through each step of the process.

By adding logic checks in `shouldComponentUpdate()` at **A** we can prevent all its children from re-rendering. This can improve overall performance significantly. But keep in mind, if you prevent **A** from passing props down to the children you may prevent required renders from occurring.

Jump ahead with `forceUpdate()`

Like `componentWillReceiveProps()`, we can skip `shouldComponentUpdate()` by calling `forceUpdate()` in the Component. This sets a flag on the Component when it gets added to the dirty queue. When flagged, `shouldComponentUpdate()` is ignored. Because we are forcing an update we are stating something has changed and the Component *must* re-render.

Since `forceUpdate()` is a brute force method, it should always be used with caution and careful consideration. You can easily get into an endless render loop if you keep triggering `forceUpdate` over and over. Troubleshooting infinite render loops can be very tricky. So, when reaching for `forceUpdate` keep all this in mind.

Next Up: Tapping into `componentWillUpdate()`

---

[1] Captured from React 15.0.1

[2] If you use Immutable.js, there is a ImmutableRenderMixin library that provides both Object shallow compare and Immutable data comparison.

Tapping into `componentWillUpdate()`

Once we have determined that we do need to re-render in our Update phase, the `componentWillUpdate()` will be called. The method is passed in two arguments: `nextProps` and `nextState`. The method `componentWillUpdate()` is similar to `componentWillMount()`, and many of the same considerations and tasks are the same. The difference being that `componentWillUpdate()` is called every time a re-render is required, such as when `this.setState()` is called. Unlike `componentWillMount()` we get access to the next `props` and `state`.

Just like `componentWillMount()`, this method is called before `render()`. Because we have not rendered yet, our Component's access to the Native UI (DOM, etc.) will reflect the old rendered UI. Unlike `componentWillMount()`, we can access `refs` but in general this is not recommended because the refs will soon be out of date. There are use cases for accessing the Native UI here, such as starting animations.

The `componentWillUpdate()` is a chance for us to handle configuration changes and prepare for the next render. If we want to access the old props or state, we can call `this.props` or `this.state`. We can then compare them to the new values and make changes/calculations as required.

Unlike `componentWillMount()`, we should not call `this.setState()` here. The reason we do not call `this.setState()` is that the method triggers another `componentWillUpdate()`. If we trigger a state change in `componentWillUpdate()` we will end up in an infinite loop [1].

Some of the more common uses for `componentWillUpdate()` is to set a variable based on state changes (not using `this.setState()`), dispatching events or starting animations [2].

```
// dispatching an action based on state change
componentWillUpdate(nextProps, nextState) {
  if (nextState.open == true && this.state.open == false) {
    this.props.onWillOpen();
  }
}
```

Up Next: Re-rendering and Children Updates

---

[1] In the previous version of this section we mistakenly said that you can safely call `setState()` in this method. Our assumption at the time was that a dirty flag was tracking the current state of the render pass, but this is not the case. It is technically possible to call

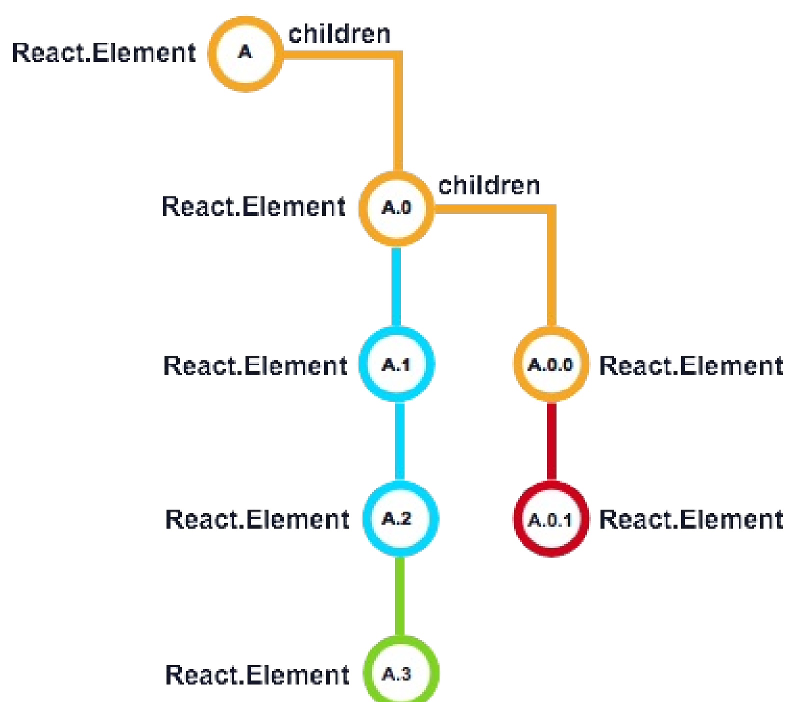
`setState()` behind a conditional (such as when a prop/state changes) but it is **not** recommended and should be considered a no go. Special thanks to Robin Venneman for catching this error and calling it to our attention!

[2] An example of triggering CSS transitions in `componentWillUpdate()` and other discussions around this method's usages is over at this StackOverflow response.

Re-rendering and Children Updates

Once again we return to `render()`. Now that our `props` and `state` are all updated[1] we can apply them to our content and children. Just like the initial render, all the same rules and conditions apply.

Unlike our first render, React performs different management when it comes to the generated Elements. The main difference is around the initialization phase and children Elements of the Component.



React compares the current Element tree structure returned from the `render()` method. React uses the generated keys (or assigned keys) to match each Element to a Component instance. React determines if we have new instances (**A.3**), removing instances (**A.0.1**) or are updating existing instances (**A**, **A.0**, **A.0.0**).

If the keys are the same, then React will pass the `props` to the existing instance, kicking off its Update life cycle. If we have added new Components or changed keys, React will create new instances from the Element data. These new Components then enter the Birth/Mounting phase.

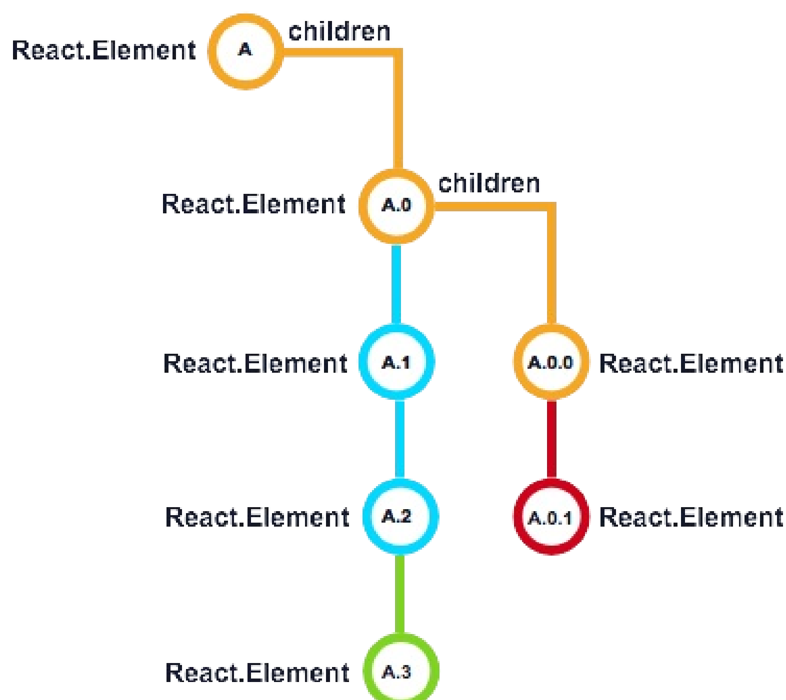
Up Next: Post-Render with `componentDidUpdate()`

[1] As mentioned earlier, the asynchronicity nature of state is now fully applied and can be accessed safely.

Post-Render with `componentDidUpdate()`

Continuing the trend of corresponding methods, the `componentDidUpdate()` is the Update version of `componentDidMount()`. Once again, we can access the Native UI stack, interact with our `refs` and if required start another re-render/update [1].

When `componentDidUpdate()` is called, two arguments are passed: `prevProps` and `prevState`. This is the inverse of `componentWillUpdate()`. The passed values are what the values were, and `this.props` and `this.state` are the current values.



Just like `componentDidMount()`, the `componentDidUpdate()` is called after all of the children are updated. Just to refresh your memory, **A.0.0** will have `componentDidUpdate()` called first, then **A.0**, then finally **A**.

Common Tasks

The most common uses of `componentDidUpdate()` is managing 3rd party UI elements and interacting with the Native UI. When using 3rd Party libraries, like our Chart example, we need to update the UI library with new data.

```
componentDidUpdate(prevProps, prevState) {
  // only update chart if the data has changed
  if (prevProps.data !== this.props.data) {
    this.chart = c3.load({
      data: this.props.data
    });
  }
}
```

Here we access our Chart instance and update it when the data has changed[2].

Another render pass?

We can also query the Native UI and get sizing, CSS styling, etc. This may require us to update our internal state or `props` for our children. If this is the case we can call `this.setState()` or `forceUpdate()` here, but this opens a lot of potential issues because it forces a new render pass.

One of the worst things to do is do an unchecked `setState()`:

```
componentDidUpdate(prevProps, prevState) {
  // BAD: DO NOT DO THIS!!!
  let height = ReactDOM.findDOMNode(this).offsetHeight;
  this.setState({ internalHeight: height });
}
```

By default, our `shouldComponentUpdate()` returns true, so if we used the above code we would fall into an infinite render loop. We would render, then call did update which sets state, triggering another render.

If you need to do something like this, then you can implement a check at `componentDidUpdate()` and/or add other checks to determine when a re-size really occurred.

```
componentDidUpdate(prevProps, prevState) {
  // One possible fix...
  let height = ReactDOM.findDOMNode(this).offsetHeight;
  if (this.state.height !== height ) {
    this.setState({ internalHeight: height });
  }
}
```

In general, this is not a common requirement and re-rendering has performance impacts for your Component and applications. Keep this in mind if you find yourself having to add a second render pass in `componentDidUpdate()`.

Up Next: Death/Unmounting In-depth

---

1 This is a risky behavior and can easily enter an infinite loop. Proceed with caution!

2

[2] This example assumes that the data is pure and not mutated.

Death/Unmount In-depth

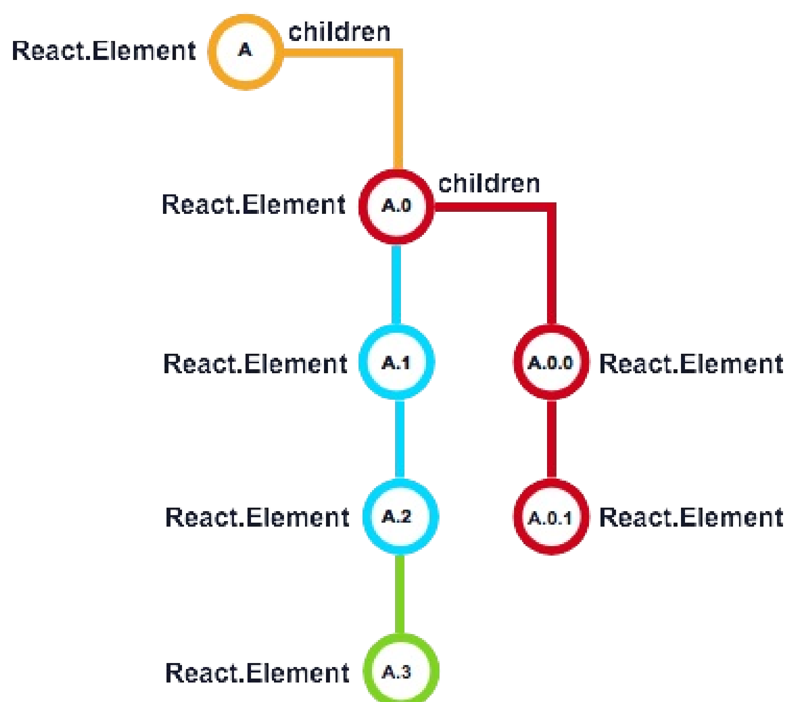
After our Component has spent time in the Update phase, we eventually enter the Death phase. During this phase our component is Unmounted from the Native UI stack and is marked for Garbage Collection.

We enter this phase when our UI changes and the Element Tree no longer has a matching key to our Component. This could be changing layout or programmatically changing keys (forcing a new Component instance to be created). Once this occurs, React looks at the instance being removed and its children.

Using `componentWillUnmount()`

Just like the rest of our life cycle phases, the Death/Unmount phase has a method hook for us. This method allows us to do some cleanup before we are removed from the UI stack. Typically we want to reverse any setup we did in either `componentWillMount()` or `componentDidMount()`.

For example, we would want to unregister any global/system/library events, destroy 3rd party UI library elements, etc. If we don't take the time to remove events we can create memory leaks in our system or leave bad references laying around.



React starts with the Element being removed, for example **A.0**, and calls `componentWillUnmount()` on it. Then React goes to the first child (**A.0.0**) and does the same, working its way down to the last child. Once all the calls have been made, React will remove the Components from the UI and ready them for Garbage Collection.

Up Next: The Life Cycle Recap

The Life Cycle Recap

We have now worked through the three phases of the React life cycle: Birth/Mounting, Growth/Update and finally Death/Unmount. By having these phases and corresponding methods React provides us a clear path for developing Components. These phases also allow us to begin to optimize our Components and our entire application.

To review, the methods and order called are:

Birth / Mounting

1. Initialize / Construction
2. `getDefaultProps()` *(React.createClass)* or `MyComponent.defaultProps` *(ES6 class)*
3. `getInitialState()` *(React.createClass)* or `this.state = ...` *(ES6 constructor)*
4. `componentWillMount()`
5. `render()`
6. Children initialization & life cycle kickoff
7. `componentDidMount()`

Growth / Update

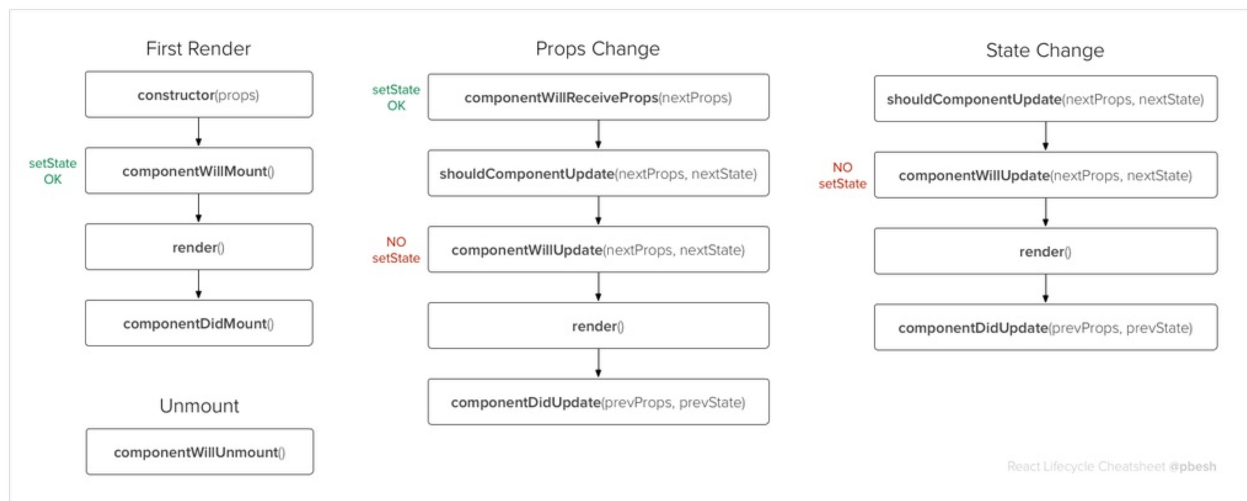
1. `componentWillReceiveProps()`
2. `shouldComponentUpdate()`
3. `componentWillUpdate()`
4. `render()`
5. Children Life cycle methods
6. `componentDidUpdate()`

Death / Un-Mounting

1. `componentWillUnmount()`
2. Children Life cycle methods
3. Instance destroyed for Garbage Collection

Life Cycle Flowchart and `setState()` safety

In addition, this flow chart by Peter Beshai breaks down the different methods and also calls out when `this.setState()` is safe and NOT safe to call:



Component Evolution and Composition

Component reuse and composability are some of the core tenets of React development. As our applications scale, development time can be dramatically reduced through this process. Yet, creating reusable Components takes planning and understanding to support multiple use cases.

Understanding the intention of the Component is the first step towards reuse. Sometimes, we know a Component will be used in many different ways from the start. In those situations, we can plan for the different scenarios right away. In other situations, Component intentions will change over the lifespan of the application. Understanding how to evolve a Component is just as important as understanding how to create reusability.

The Application Architecture process

Let's take a quick moment and discuss the process of application architecture. We often hear about over-architected systems. This often occurs when we try to plan for every possible scenario that could ever occur through the life of an application. To try and support every conceivable use is a fools errand. When we try to build these systems we add unnecessary complexity and often make development harder, rather then easier.

At the same time, we don't want to build a system that offers no flexibility at all. It may be faster to just build it without future thought, but adding new features can be just as time consuming later on. Trying to find the right balance is the hardest part of application architecture. We want to create a flexible application that allows growth but we don't want to waste time on all possibilities.

The other challenge with application architecture is trying to understand our needs. With development, we often have to build out something to truly understand it. This means that our application architecture is a living process. It changes over time due to having a better understanding of what's required. Refactoring Components is critical to the success of a project and makes adding new features easier.

Because of this process, we felt it is important to walk through the evolution of a Component. We will start with a naive approach to building a List Component and then walk through different refactorings to support reusability. More then likely, we would know early on that a List should be reusable. But, walking through the evolution process can help deepen our understanding of how to enable reusability.

Up Next: The Evolution of a List Component

The Evolution of a List Component

Lists are everywhere in applications today. The list is crucial to Social Media UIs, such as Facebook, Twitter, Reddit, Instagram, etc. The current demo app trend of Todos are all about displaying a list of items. The lowly HTML drop-down displays a list of selectable options. It's so common, most of us take lists for granted.

When we start building our application, how should we approach creating reusable Components? Let's walk through a possible progression of a list feature.

The first pass

Typically, the first approach is to build a React component that renders the UI to the specific layout and data needs. For our example, we are building a list of customer profiles. The first design round requires the profile to have an avatar/picture and descriptive text.

UI Wireframe #1



The first step would be to create a Component that takes an Array of Objects, which has an image path and the description text. Our Component will loop over this Array and render out each element, using `<li>` items.

```
import React from 'react';

class List extends React.Component {
  renderProfiles () {
    return this.props.profile.map( (profile) => {
      return (
        <li>
          <img href={ profile.imagePath } align="left" width="30" height="30" />
          <div className="profile-description">
            { profile.description }
          </div>
        </li>
      );
    });
  }

  render() {
    return (<ul className="profile-list">{ this.renderProfiles() }</ul>);
  }
}

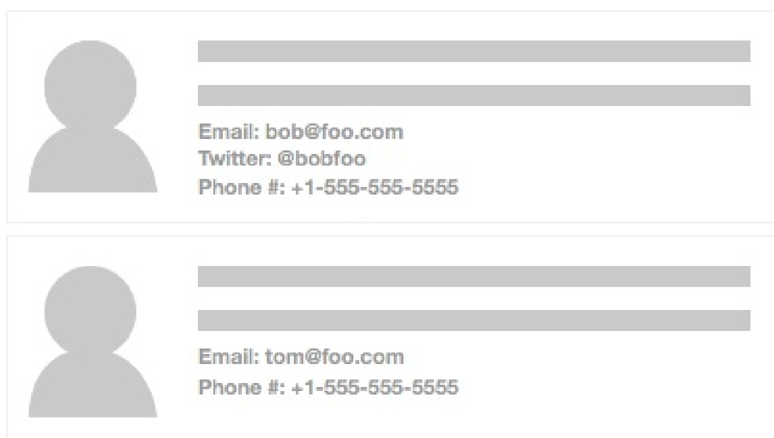
List.defaultProps = { profile: [] };
export default List;
```

We would then apply styling to the `<ul>`, `<li>` and `<div>` elements to meet our design needs. This Component is a simple way of rendering out our content. It meets our design needs but isn't reusable.

Requirements change

As with any project, needs change. For our example, the users now want to list more details about each customer. The design team comes up with a new layout and we now have to support optional fields.

UI Wireframe #2



With this new design we now need to do our first bit of Component refactoring. To support the new optional detail fields, we need to add logic to our Profile rendering. A good development practice is to keep our React Components as compartmentalized as possible. This enables multiple benefits.

First, it helps reduce cognitive load. Having smaller, single focused Components means they are easier to read and understand the intention. A common experience we have all had as developers is returning to our own code six or more months later. Because we wrote it, we *should* understand it, but often it takes a bit of time to put ourselves back into mindset of what the code is solving. If the Component has hundreds of lines of logic, it will take that much more time to grok what the intention is. Even harder (and time consuming) is doing this with another developer's work.

One of the beautiful features of React is that we can (and should) break our Components into small bite-sized chunks. Because it is so easy in React, this helps us make our code easier to understand. At the same time, this leads to the second benefit: faster reusability.

If we break out a Component to a single task, such as rendering a single profile, we now have the potential to reuse it. It is possible that elsewhere in the app we need to show a profile. With our current implementation, this is not easily done. This is because the rendering of the profile details is handled internally by the List component. Let's break the profile details out into a new Component and refactor our List a bit.

Creating a Profile Component

The first step is to move the render code from the List into it's own Component.

Profile.js

```
import React from 'react';

export default class Profile extends React.Component {
  renderDetails(key, label){
    if (this.props[key]) {
      return (<div className="detail">{ label } { this.props[key] }</div>);
    }
  }

  render() {
    return (
      <li>
        <img href={ this.props.imagePath } align="left" width="30" height="30" />
        <div className="profile-description">
          { this.props.description }
        </div>
        { this.renderDetails('email', 'Email:') }
        { this.renderDetails('twitter', 'Twitter:') }
        { this.renderDetails('phone', 'Phone:') }
      </li>
    );
  }
}
```

Here we have broken out the optional details rendering into a new Component called `Profile`. Profile's job is to render out the base layout and then render out our optional details, depending on if they are defined or not[1]. We can then update our List code:

List.js

```
import React from 'react';
import Profile from './Profile';

class List extends React.Component {
  render() {
    return (
      <ul>
        { this.props.profile.map( (profile) => <Profile {...profile} /> ) }
      </ul>
    );
  }
}

List.defaultProps = { profile: [] };
export default List;
```

Now our List maps the profile data and sends it to the `Profile` Component for rendering. By isolating the rendering of the profile to a single component we have a clear separation of concerns (SoC). Not only do we get the benefit of SoC, we also make each Component a lot easier to understand. When we have to return to this code six months later, it will be a lot faster to get caught back up.

Up Next: Rendering Different Content

---

[1] Following this pattern we could go even further if so desired. We could break out each Profile detail into its own Component. Yet, that maybe going too far down the granularity rabbit hole. Once again, over-architecture is a slippery slope and having to make a judgment call is part of the process.

Rendering different content

By moving our UI rendering of each Profile to a Component, we have separated layout and content display. The List is responsible for layout and data management. The Profile is responsible for UI rendering for each individual item. Because of this first step, we can move one step further and make our List even more flexible.

List Feature expansion

Continuing our customer example, let's imagine that our Profile List has started to evolve even more. We have added pagination support, selection management, sorting, filtering, etc. Now, our users request the ability to manage a different kind of content. They now want to manage Posts.

These Posts have some similar UI elements as our profile: images, descriptions, and details. But the layout and content vary drastically. We still need all the of the functionality of the List; pagination, filtering, etc. The question becomes, how do we handle this?

Item Rendering

A simple, but not ideal, approach would be to add a switch in our List's `map` method. The switch checks the data type and then chooses to use the Profile Component or the Post Component. But, this approach adds a pretty bad code smell. Similar to our first draft of the List, it meets our immediate needs but what happens when we need a Message List? Or Viewer List? Soon our List has a lot of switches.

A better way to solve this is through configuration. We can expose a prop on the List component that handles the rendering of each item. There are two ways to do this: by passing in a function or by passing in a Component Class.

Function Item Renderer

The first approach we will examine is passing in a function that handles rendering out each individual item in the List. The first step is to update our List component to require a `itemRenderer` prop that is a function and changing our profiles `prop` to items.

List.js

```
import React from 'react';

class List extends React.Component {
  render() {
    return (
      <ul>
        { this.props.items.map( (item, index) => this.props.itemRenderer(item, index)
) }
      </ul>
    );
  }
}

List.propTypes = {
  items: React.PropTypes.array,
  itemRenderer: React.PropTypes.func.isRequired
};
List.defaultProps = { items: [] };
export default List;
```

We have added a `propTypes` configuration to require the `itemRenderer` prop, which needs to be a function. We also added an items `prop` , which replaces `profiles` . In our `render()` we now call the function passing in the item instance data and the index. We will talk more about why we need to pass `index` in a bit. In our parent Component or App we now do the following:

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import List from './components/List';
import Profile from './components/Profile';
import Posts from './components/Posts';

let profileData = [ ... ] // psuedo code, this has all our profile data
let postsData = [ ... ] // psuedo code, this has all our post data

class App extends React.Component {
  renderProfile(profile, key) {
    return (<Profile {...profile} key={ key } />);
  }

  renderPosts(posts, key) {
    return (<Posts {...post} key={ key } />);
  }

  render() {
    return (
      <div>
        <List items={ profileData } itemRenderer={ this.renderProfile } />
        <List items={ postsData } itemRenderer={ this.renderPosts } />
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('mount-point'));
```

In `index.js` we render out two different List components. For the first List, we pass in our profile data and our `renderProfile` method reference. Just like any React action (such as `onClick`) we pass the method reference and do not actually call the method. For the second, we pass in the posts data and the `renderPosts` method reference.

When the Lists render, the `map` method calls either `renderProfile()` or `renderPosts()` with each data element and the current index.

React keys and arrays of components

The reason we pass index is that we need to generate a unique key for each item in the list. When we offload rendering to a method, we no longer get React's built in ability to generate the keys for us.

React Component keys are used for Component Reconciliation:

> Reconciliation is the process by which React updates the DOM with each new render pass...
>
> ... The situation gets more complicated when the children are shuffled around (as in search results) or if new components are added onto the front of the list (as in streams). In these cases where the identity and state of each child must be maintained across render passes, you can uniquely identify each child by assigning it a key
>
> When React reconciles the keyed children, it will ensure that any child with key will be reordered (instead of clobbered) or destroyed (instead of reused).
>
> -- React Child Reconciliation

If we don't set a key when generating children dynamically (via our `itemRenderer` method) we would get the following warning:

> Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `List`. See https://fb.me/react-warning-keys for more information.

The quick solution is to pass in the index of the data, but this may not be the ideal solution. The problem with this approach is that it generates a key based on item order. It would be better to use a unique `id` that's defined on the data set. Another option is generating a hash code or some other unique identifier that reflects the data element's content.

By having an identifier based on the data content instead of order, we can help optimize the Component rendering. When we display partial lists, such as filtering or sorting, if our key is based on the content and not order, React knows it doesn't have to generate a new Element for the data. It just needs to reorder the elements.

Component Item Renderer

Another option for handling dynamic renderers, is to use a Component Class reference. This process is similar to passing in a function. Instead of offloading the rendering to the return value of a method we create a React Element from the Component and pass in the configuration.

List.js

```
import React from 'react';
import Profile from './Profile';

class List extends React.Component {
  render() {
    return (
      <ul>
        { this.props.profile.map( (profile, index) => {
            let newProps = Object.assign({ key: index }, profile);
            return React.createElement(this.props.itemRenderer, newProps);
        }) }
      </ul>
    );
  }
}

List.propTypes = { itemRenderer: React.PropTypes.func };
List.defaultProps = { profile: [], itemRenderer: Profile };
export default List;
```

In this version of the List Component, we create a new React Element using the `this.props.itemRenderer` as the Component Class type. We generate a `newProps` object that adds the `key` to the profile data and pass this to the Element as its `props`.

Because we define a default item renderer of `Profile` in the `defaultProps` we can update `propTypes` to make `itemRenderer` an optional param. To use this version of the List our index.js now looks like this:

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import List from './components/List';
import Profile from './components/Profile';
import Post from './components/Post';

let profileData = [ ... ] // psuedo code, this has all our profile data
let postsData = [ ... ] // psuedo code, this has all our post data

class App extends React.Component {
  render() {
    return (
      <div>
        <List items={ profileData } />
        <List items={ postsData } itemRenderer={ Post } />
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('mount-point'));
```

Since we have a default item renderer (the Profile Component), the first version of the List just needs the profile data. In the second version, we change out the renderer type by passing in our Component and passing in the item data.

When our List renders the data it now creates a React Element from the `itemRenderer` value and passes in the current data element. At DevelopmentArc, we have found using a React Class is a much cleaner approach to developing replaceable UI elements.

Up Next: Higher Order Components

Higher Order Components

The last Component composition pattern we will examine in this section is *Higher Order Components* (HOC). As Dan Abramov discusses, Higher Order Components were first proposed by Sebastian Markbåge in a gist. The core idea of HOC is to define a function, which you pass one or more Components to. This function generates and returns a new Component, which is a wrapper around the passed in Component(s).

The need for HOC came about with React's move to support ES6 classes and the lack of mixin support with the new JavaScript Class syntax. To handle this change, a new pattern needed to be defined to replace mixins. Typically, mixins add/override functionality around the Component Life Cycle and enable sharing reusable code in a elegant way. Without mixin support in ES6, the HOC pattern is required.

A form group example

For our HOC example, we will create a function for wrapping a Component in a custom form group with an optional `<label>` field. The goal of the HOC is to allow us to create two outputs, with and without a label:

```
<!-- With a label -->
<div class="form-group">
  <label class="form-label" for="firstName">First Name:</label>
  <input type="text" name="firstName" />
</div>

<!-- Without a label -->
<div class="form-group">
  <input type="text" name="lastName" />
</div>
```

Because this could become a common task, we can use the HOC pattern to generate our form group wrapper and let it decide if it should inject the label or not.

formGroup.js

```
import React from 'react';
import { isString } from 'lodash';

function formGroup(Component, config) {
  const FormGroup = React.createClass({
    __renderLabel() {
      // check if the passed value is a string using Lodash#isString
      if (isString(this.props.label)) {
        return(
          <label className="form-label" htmlFor={ this.props.name }>
            { this.props.label }
          </label>
        );
      }
    },

    __renderElement() {
      // We need to see if we passed a Component or an Element
      // such as Profile vs. <input type="text" />
      if (React.isValidElement(Component)) return React.cloneElement(Component, this.p
rops);
      return( <Component { ...this.props } />);
    },

    render() {
      return(
        <div className="form-group">
          { this.__renderLabel() }
          { this.__renderElement() }
        </div>
      );
    }
  });

  return(<FormGroup { ...config } />);
}

export default formGroup;
```

To use this HOC we can do the following:

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import formGroup from './higherOrderComponents/formGroup';

let MyComponent = React.createClass({
  render() {
    return (
      <div>
        { formGroup(<input type="text" />, { label: 'First Name:', name: 'firstName' }
) }
      </div>
    );
  }
});

ReactDOM.render(<MyComponent />, document.getElementById('mount-point'));
```

Let's examine the above code. The first thing we do for the HOC is create a function called `formGroup` which takes two arguments: `Component` and `config`.

```
function formGroup(Component, config) {
  ...
}

export default formGroup;
```

The Component will be the instance we want to wrap in our form group. In the function, we create a new React Component and then return an Element instance using the `config` as props.

```
const FormGroup = React.createClass({
  ...
});

return(<FormGroup { ...config } />);
```

We take advantage of the ES6 spread operator to pass in our `config` object as the props for the generated JSX Element. In our `render()` method we create the form group `<div>` and then render out our optional label and Component content.

```
render() {
  return(
    <div className="form-group">
      { this.__renderLabel() }
      { this.__renderElement() }
    </div>
  );
}
```

In our `__renderLabel()` method[1] we use the Lodash `isString` method to check if the label value is a string. If so, we render out our label DOM element, otherwise we return `null`.

```
__renderLabel() {
  // check if the passed value is a string using Lodash#isString
  if (isString(this.props.label)) {
    return(
      <label className="form-label" htmlFor={ this.props.name }>
        { this.props.label }
      </label>
    );
  }
},
```

Because `null` does not render out to the Native UI in React, this is how we make the `<label>` optional based on the passed value.

Finally, we had to add a check to determine what type was passed to our HOC function for the Component. This is an important check because we want to support both React Components and Elements.

In our `index.js` we are passing in:

```
formGroup(<input type="text" />, { label: 'First Name:', name: 'firstName' })
```

Because we are using JSX to generate our `<input />` the HOC will receive an Element. But, if we used our Profile component, we may not want to use JSX:

```
formGroup(Profile, { label: 'First Name:', name: 'firstName' })
```

To support both options and pass on the `props`, we use the `__renderElement()` method to handle the inspection and output generation:

```
__renderElement() {
  // We need to see if we passed a Component or an Element
  // such as Profile vs. <input type="text" />
  if (React.isValidElement(Component)) return React.cloneElement(Component, this.props
);
  return( <Component { ...this.props } />);
},
```

If the Component instance is an element, we clone the element and pass on the new props. Otherwise, we generate a new Element using JSX and the passed in React Component.

This HOC example is just the tip of the iceberg when it comes to self-generating wrapper components. Using this pattern, we can tap into the Component Life Cycle methods, we can make more complex decisions based on the data, we can register to stores or other events, and many other possible combinations.

For more in-depth examples we highly recommend reading Dan Abramov's *Mixins Are Dead. Long Live Composition* and @franlplant's *React Higher Order Components in depth*

---

[1] In these examples we are prefixing our methods with `__` to reflect that these are internal component methods. This is completely optional and is just our preferred style syntax.

About the Authors

James Polanco

@jamespolanco

In his role as co-founder and CTO of DevelopmentArc®, James strives to take powerful business ideas and integrate them into elegant technological experiences to help clients achieve their overall strategic goals. Since 1996, James has helped plan, guide and create interactive and web-based solutions for companies including Adobe, Toyota Motor Sports, BlueKai, VMWare, Macromedia, and DHAP Digital. James is also an international speaker, presenting on technology implementations & processes and a published author on the topic of full team product development and company workflows.

Aaron Pedersen

@aaronpedersen

As co-founder and CEO of DevelopmentArc®, a boutique development firm and parent company of Pedanco, Aaron Pedersen's passion lies in helping businesses streamline process making teams work more effectively through innovative technology solutions. A published author, expert speaker, and sought-after business consultant and trainer, Aaron works with a wide range of companies, from Fortune 500 corporations and multi-chain hospitality companies to emerging brands and seed-round startups including Toyota Motor Sports, DHAP Digital, Adobe, KitchenNetwork and FitStar.